On Automation of CTL* Verification for Infinite-State Systems

Byron Cook[1], Heidy Khlaaf[1], and Nir Piterman[2]

[1] University College London
[2] University of Leicester

Abstract. In this paper we introduce the first known fully automated tool for symbolically proving CTL^* properties of (infinite-state) integer programs. The method uses an internal encoding which facilitates reasoning about the subtle interplay between the nesting of path and state temporal operators that occurs within CTL^* proofs. A precondition synthesis strategy is then used over a program transformation which trades nondeterminism in the transition relation for nondeterminism explicit in variables predicting future outcomes when necessary. We show the viability of our approach in practice using examples drawn from device drivers and various industrial examples.

1 Introduction

In recent years, a number of systems have been proposed to automate the verification of either branching-time properties (e.g. expressed in CTL) or linear-time properties (e.g. LTL) of general integer manipulating programs [3, 11, 9, 7, 10]. Branching-time property verification requires reasoning about sets of *states* within a transition system that satisfy a particular temporal formula. Contrarily, linear-time property verification requires reasoning about sets of *paths* that satisfy a formula. However, these logics have significantly reduced expressiveness as they restrict or disallow the interplay between linear-time and branching-time operators. For example, a property involving the assertion "along *some* future an event occurs *infinitely often*" cannot be expressed in either LTL nor CTL, yet is crucial when expressing the existence of fair paths spawning from every reachable state in an infinite-state system. Contrarily, CTL^* is capable of expressing CTL, LTL, and properties necessitating their interplay, as demonstrated by examples further below.

Unfortunately, no fully automatic CTL^* proving methods for infinite-state systems are known. Despite the existence of automated verification tools for branching-time and linear-time temporal logic, these tools do not allow for the verification of CTL^*. A key problem is that CTL^* formulae cannot merely be partitioned into isolated CTL and LTL sub-formulae, as such a partition fails to treat the intricate dependence between state-based and path-based reasoning. In this paper we introduce the first known automatic method capable of proving CTL^* properties of infinite-state programs. Our contribution is a method that allows for the arbitrary nesting of state-based reasoning within path-based reasoning, and vice versa. Towards this purpose we recursively deconstruct a CTL^* formula in a way that allows us to determine where the subtle interplay between the arbitrary nesting of path and state formulae occurs. To reason about the path subformulae, we find a sufficient set of branching nondeterministic decisions within a program's transition relation. We then devise a method of *temporarily* substituting said nondeterministic decisions with a *partially symbolic determinized* form. That is, nondeterministic decisions regarding which paths are taken are determined by variables which summarize the future of the program execution. When interchanging between path and state formulae, these determinized relations must then be collapsed to incorporate path quantifiers. Preconditions for the given CTL^* property can then be acquired via existing CTL model checkers.

Based on our approach, we have developed a tool capable of automatically proving properties of programs that no tool could previously fully automate. The paper closes with a description of our experimental results using the developed tool on various programs drawn from industrial examples. Our tool is available under the MIT open-source license at `https://github.com/hkhlaaf/T2/tree/T2Star`.

Expressiveness of CTL^*. CTL^* allows us to express properties involving existential system stabilization, stating that an event can eventually become true and stay true from every reachable state. Additionally, it can express "possibility" properties, such as the viability of a system, stating that every reachable state can spawn a fair computation. Below are properties that can only be afforded by the extra expressive power of CTL^*. These liveness properties are often imperative to verifying systems such as Windows kernel APIs that acquire resources and APIs that release resources, as later shown by our experiments.

For example, the property $\mathsf{EFG}(\neg x \wedge (\mathsf{EGF}\ x))$ conveys the divergence of paths. That is, there is a path in which a system stabilizes to $\neg x$, but every point on said path has a diverging path in which x holds infinitely often. This property is not expressible in CTL or in LTL, yet is crucial when expressing the existence of fair paths spawning from every reachable state in a system. In CTL, one can only examine sets of states, disallowing us to convey properties regarding paths. In LTL, one cannot approximate a solution by trying to *disprove* either $\mathsf{FG}\ \neg x$ or $\mathsf{GF}\ x$, as one cannot characterize these proofs within a path quantifier.

Another CTL^* property $\mathsf{AG}\big[(\mathsf{EG}\ \neg x) \vee (\mathsf{EFG}\ y)\big]$ dictates that from every state of a program, there exists either a computation in which x never holds or a computation in which y eventually always holds. The linear time property $\mathsf{G}(\mathsf{F}x \rightarrow \mathsf{FG}\ y)$ is significantly stricter as it requires that on every computation either the first disjunct or the second disjunct hold. Finally, the property $\mathsf{EFG}\big[(x \vee (\mathsf{AF}\ \neg y))\big]$ asserts that there exists a computation in which whenever x does not hold, all possible futures of a system lead to the falsification of y. This assertion is impossible to express in LTL.

Related work. Proof systems for the verification of CTL^*, first introduced by [13, 20], have been well-studied. It is known that CTL^* model checking for infinite-state systems generalizes termination and co-termination and is undecidable. A decision procedure exploring the structure of finite-state ω-automata was first introduced to determine the satisfaction of a CTL^* formula over binary relations in [16], and later extended in [14]. A complete and sound axiomatization of propositional CTL^* then followed in [25], which inspired the first sound and relatively complete deductive proof system for the verification of CTL^* properties over possibly infinite-state reactive systems [19]. Proof rules for verifying CTL^* properties of infinite-state systems were implemented in STeP [4]. However, the STeP system is only semi-automated, as it still requires users to construct auxiliary assertions and participate in the search for a proof.

Model checking CTL^* [15] for finite-state programs and other decidable settings has been implemented in [17]. Their approach reduces a CTL^* formula to μ-calculus using a system of fixed-point equations on relations with first-order quantifiers and equalities. They then invoke a μ-calculus model checker. Contrarily, we seek to verify the undecidable general class of infinite-state programs supporting both control-sensitive and integer properties. Given that μ-calculus model checking is polynomial-time equivalent to the solution of parity games [14], one can conceive that the approach in [2] could potentially solve CTL^* model checking if the latter were reduced to solving parity games by combining [17] and [14]. However, we note that the resulting infinite-state game would integrate the (first-order μ-calculus) property within the program making it difficult to extract invariants pertaining the program. For this reason, it is often the case that such a series of reductions inhibits tool performance. Furthermore, [2] requires a manual instantiation of the structure of assertions, characterizing subsets of the infinite-state game, that are to be found by their tool.

Existing automated tools for verification of infinite-state programs support *either* branching-time only *or* linear-time only reasoning, e.g., [5, 9, 11, 7, 3, 10, 26]. The important distinction however is that these tools do not allow for the interaction between linear-time and branching-time formulae.

Finally, we have adopted and repurposed a similar symbolic determinization technique introduced in [11] for the verification of LTL formulae in the infinite-state setting. Their symbolic determinization is based on the counterexample-guided refinement of generated tree counterexamples, or counterexamples with branching paths. That is, [7] produce a semantics-preserving transformation that encodes the structure of the nested CTL formulae within the state space, allowing for the generation of tree counterexamples. This causes precondition generation for syntactically partitioned formulae to be no longer possible, limiting the interplay between linear-time operators and path quantifiers allowed by our strategy.

Limitations. Our tool does not support programs with heap, nor do we support recursion or concurrency. The heap-based programs we consider during our experimental evaluation have been abstracted using an over-approximation technique introduced by [21]. Effective techniques for proving temporal properties of programs with heap remains an open research question. Our technique relies on the availability of CTL model checking and non-termination procedures. It is, in principle, applicable to every class of infinite-state systems for which such procedures are available (provided that integer variables are allowed). Additionally, our procedure is not complete as we use a series of techniques for safety [23], termination [24, 8], nontermination [18], and CTL [3, 10] that are not complete. Furthermore, our determinization procedure is not complete. We will further address this issue in later sections.

2 Preliminaries

Programs. As is standard [22], we treat programs as control-flow graphs, where edges are annotated by the updates they perform to variables. A program is a triple $P = (\mathcal{L}, E, \mathsf{Vars})$, where $\mathcal{L}$ is a set of locations, E is a set of edges/transitions, and Vars is a set of variables. Each edge $\tau = (\ell, \rho, \ell')$ in E, where $\ell, \ell' \in \mathcal{L}$ and ρ is a condition, specifies possible transitions in the program. The condition ρ is an assertion in terms of Vars and Vars', a primed copy of Vars, where constants range over Vals. That is, Vars refers to the values of variables before an update and Vars' refers to the values of variables after an update.

The set of locations includes the first location $\ell_{_I}$, which has no incoming transitions from other program locations. That is, for every $\tau = (\ell, \rho, \ell') \in E$ we have $\ell' \neq \ell_{_I}$. Transitions exiting $\ell_{_I}$ have their conditions expressed in terms of Vars'. Locations with incoming transitions from $\ell_{_I}$ are *initial locations*. This allows us to encode more complex initial conditions. In figures, we omit $\ell_{_I}$ and merely display the edges to locations with incoming transitions from $\ell_{_I}$.

A program gives rise to a transition system $T = (S, R)$, where S is the set of program states of the form $S = (\mathcal{L} - \{\ell_{_I}\}) \times (\mathsf{Vars} \rightarrow \mathsf{Vals})$ and $R \subseteq S \times S$. That is, a program state is a pair (ℓ, f) where $\ell \neq \ell_{_I}$ and f is a valuation, i.e., a function from program variables to values. A program can transition from (ℓ, f_1) to (ℓ', f_2) if there exists a transition $(\ell, \rho, \ell') \in E$ such that $(f_1, f_2) \models \rho$. The valuation (f_1, f_2) is a function from $\mathsf{Vars} \cup \mathsf{Vars}'$ to Vals such that for every $v \in \mathsf{Vars}$, $(f_1, f_2)(v) = f_1(v)$ and $(f_1, f_2)(v') = f_2(v)$. A state (ℓ, f) is considered initial if there is a transition $(\ell_{_I}, \rho, \ell)$ such that $(f_{-1}, f) \models \rho$, where f_{-1} is some arbitrary valuation. Notice that ρ is expressed in terms of Vars' and hence the valuation f_{-1} does not affect the satisfaction of ρ.

Given $V \subseteq \mathsf{Vars}$, the valuation obtained from f by restricting the valuation to variables in V is denoted by $f{\Downarrow_V}$. The restriction of states of the form (ℓ, f) and paths in the program is defined similarly, e.g., $\pi{\Downarrow_V}$.

Paths. A *path* or a *trace* π in P is an infinite sequence of states $(\ell_0, f_0), (\ell_1, f_1), \ldots$, where for every $i \geq 0$, there exists some $(\ell_i, \rho_i, \ell_{i+1}) \in E$ where $(f_i, f_{i+1}) \models \rho_i$. We say that π is an (ℓ, f)-path if $\ell_0 = \ell$ and $f_0 = f$. Given a program P, a location ℓ, and a valuation f, we denote the set of (ℓ, f)-paths in P by $\mathsf{Path}(P, \ell, f)$. We say that π is a computation in P if (ℓ, f) is initial. Note that we restrict our attention to infinite paths and computations. In practice, we modify programs, transition systems, and temporal logic formulae to ensure that all paths are infinite, as is done, e.g., in [6].

CTL^*. We are interested in verifying full computation tree logic (CTL^*) [20, 13]. The syntax of CTL^* (written in negation normal form) includes state formulae φ, that are interpreted over states, and path formulae ψ, that are interpreted over paths. We assume that atomic propositions (ranged over by α) are expressed in some underlying theory over variables and constants (*e.g.* $\mathsf{x} < \mathsf{y}$). State formulas (φ) and path formulas (ψ) are co-defined:

$$\begin{aligned} \varphi &::= \alpha \mid \neg\alpha \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{A}\psi \mid \mathsf{E}\psi \\ \psi &::= \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{G}\psi \mid \mathsf{F}\psi \mid [\psi \mathsf{W} \psi] \mid [\psi \mathsf{U} \psi] \end{aligned}$$

For a program P and a CTL^* state formula φ, we say that φ holds at a state s in P, denoted by $P, s \models \varphi$ if:

– If $\varphi = \alpha$, then $P, s \models \alpha$ iff $s \models \alpha$
– If $\varphi = \neg\alpha$, then $P, s \models \neg\alpha$ iff $s \not\models \alpha$
– If $\varphi = \varphi_1 \vee \varphi_2$, then $P, s \models \varphi_1 \vee \varphi_2$ iff $s \models \varphi_1$ or $s \models \varphi_2$
– If $\varphi = \varphi_1 \wedge \varphi_2$, then $P, s \models \varphi_1 \wedge \varphi_2$ iff $s \models \varphi_1$ and $s \models \varphi_2$
– If $\varphi = \mathsf{A}\psi$, then $P, s \models \mathsf{A}\psi$ iff $\forall \pi = (s, ...).\ P, \pi \models \psi$
– If $\varphi = \mathsf{E}\psi$, then $P, s \models \mathsf{E}\psi$ iff $\exists \pi = (s, ...).\ P, \pi \models \psi$

Path formulae are interpreted over paths. For a program P and a CTL^* path formula ψ, we say that ψ holds on a path $\pi = (s_0, s_1, \ldots)$ in P for location i, denoted by $P, \pi, i \models \psi$ if:

– If $\psi = \varphi$ is a state formula, then $P, \pi, i \models \varphi$ iff $P, s_i \models \varphi$.
– If $\psi = \psi_1 \vee \psi_2$, then $P, \pi, i \models \psi_1 \vee \psi_2$ iff $P, \pi, i \models \psi_1$ or $P, \pi, i \models \psi_2$
– If $\psi = \psi_1 \wedge \psi_2$, then $P, \pi, i \models \psi_1 \wedge \psi_2$ iff $P, \pi, i \models \psi_1$ and $P, \pi, i \models \psi_2$
– If $\psi = \mathsf{F}\psi_1$, then $P, \pi, i \models \mathsf{F}\psi_1$ iff $\exists j \geq i.\ P, \pi, j \models \psi_1$
– If $\psi = \mathsf{G}\psi_1$, then $P, \pi, i \models \mathsf{G}\psi_1$ iff $\forall j \geq i.\ P, \pi, j \models \psi_1$
– If $\psi = \psi_1 \mathsf{W} \psi_2$, then $P, \pi, i \models \psi_1 \mathsf{W} \psi_2$ iff either $\exists k \geq i.\ P, \pi, k \models \psi_2$ and $\forall i \leq j < k.\ P, \pi, j \models \psi_1$ or $\forall j \geq i.\ P, \pi, j \models \psi_1$
– If $\psi = \psi_1 \mathsf{U} \psi_2$, then $P, \pi, i \models \psi_1 \mathsf{U} \psi_2$ iff $\exists k \geq i.\ P, \pi, k \models \psi_2$ and $\forall i \leq j < k.\ P, \pi, j \models \psi_1$

A path formula ψ holds in a path π, denoted by $P, \pi \models \psi$, if $P, \pi, 0 \models \psi$. For a state formula φ, φ holds on P, denoted by $P \models \varphi$, if for every initial state s we have $P, s \models \varphi$. When the program P is is clear from the context, we may write $s \models \varphi$ for a state formula φ or $\pi, i \models \psi$ for a path formula ψ.

The branching-time logic CTL is a restricted subset of CTL^* in which temporal operators cannot be nested. That is, the only path formulas allowed are $\mathsf{G}\varphi_1$, $\mathsf{F}\varphi_1$, $\varphi_1 \mathsf{U} \varphi_2$, and $\varphi_1 \mathsf{W} \varphi_2$ for state formulas φ_1 and φ_2. The linear-time logic LTL is a fragment of CTL^* that only allows formulae of the form $\mathsf{A}\psi$, where A is the only occurrence of a path quantifier within ψ. When taking LTL as subset of CTL^*, LTL formulae are implicitly prefixed with the universal path quantifier A.

Strongly connected subgraphs. We provide some notation regarding strongly-connected subgraphs followed by the definition of *relation pairs* below. For a program P, we denote an ordered sequence of locations $\ell_0, ..., \ell_n$ as a cycle c if $\ell_n = \ell_0$ and for every $i \geq 0$ there exists some $(\ell_i, \rho_i, \ell_{i+1}) \in E$. Let C be the set of program locations such that $\ell \in \mathcal{L}$ appears in a cycle c. That is, $C = \{\ell \mid \exists c.\ \ell \in c\}$. For a program P and the set of locations C, we identify $\mathrm{SCS}(P, C)$ as some maximal set of non-trivial strongly-connected subgraphs (SCSs) of P such that every two subgraphs $G_1, G_2 \in \mathrm{SCS}(P, C)$ are either disjoint or one is contained in the other and for every $\ell \in C$, there exists at least one

$G \in \mathrm{SCS}(P, C)$ such that $\ell \in G$. The details regarding the identification of C and $\mathrm{SCS}(P, C)$ are standard and thus omitted here (see, e.g., [12]). We denote the minimal SCS in $\mathrm{SCS}(P, C)$ that contains a location $\ell \in \mathcal{L}$ by $\textsc{MinSCS}(P, C, \ell)$.

Identifying a program's strongly-connected subgraphs allows us to sufficiently find the set of *relation pairs* that characterize instances of branching nondeterministic decisions within a program's transition relation. A relation pair is thus (ρ_1, ρ_2) such that for some location ℓ we have (ℓ, ρ_1, ℓ_1) and (ℓ, ρ_2, ℓ_2) are transitions of P and $\ell_1 \in \textsc{MinSCS}(P, C, \ell)$ and $\ell_2 \notin \textsc{MinSCS}(P, C, \ell)$. That is, ρ_1 is the condition for remaining in the (minimal) SCS of ℓ and ρ_2 is the condition for leaving the (minimal) SCS of ℓ.

3 Overview

In this section, we present a quick overview of our CTL^* verification procedure $\textsc{ProveCTL}^*$, presented in Fig. 3 with an in-depth explanation provided later in Section 4. The procedure is designed to recurse over the structure of a given CTL^* formula, and for each sub-formula θ we produce a precondition a that ensures its satisfaction. That is, a is an assertion over program variables and locations characterizing the states of the program that satisfy θ. We start by finding the precondition of the innermost sub-formula, followed by searching for the preconditions of the outer sub-formulae dependent on it.

A given CTL^* formula is deconstructed to differentiate between state and path sub-formulae, as the crux of verifying CTL^* formulae lies within identifying the interplay between the arbitrary nesting of path and state formulae. Preconditions for branching-time logic state formulae can be acquired via existing CTL model checking techniques which return an assertion characterizing the states in which a sub-formula holds. The essence of our algorithm is thus within how we acquire sufficient preconditions for path formulae that admit a sound interaction with state formulae. The algorithm is based on the procedures below, which are defined in later sections of the paper:

$\textsc{Approximate}$ is a procedure that performs a syntactic conversion from a path formula to its corresponding over-approximated universal CTL formula (ACTL)[1]. The over-approximated formula can then be checked by an existing CTL model checker over a partially symbolic determinized form of the program to reduce path formula verification to state formula verification.

$\textsc{Determinize}$ allows us to reason about path characterization through state characterization, as the satisfaction of an ACTL over-approximated formula implies the satisfaction of the path formula. However, the inverse does not hold. The procedure thus constructs a form of a partially determinized program over the symbolic representations of all characterized instances of branching nondeterminism (i.e. *relation pairs*), stemming from the same program location ℓ. That

[1] ACTL is the universal subset of CTL where one can only address all possible paths with the universal quantifier A (e.g. AG or AF), but not the existence of some paths with E (e.g. EG or EF).

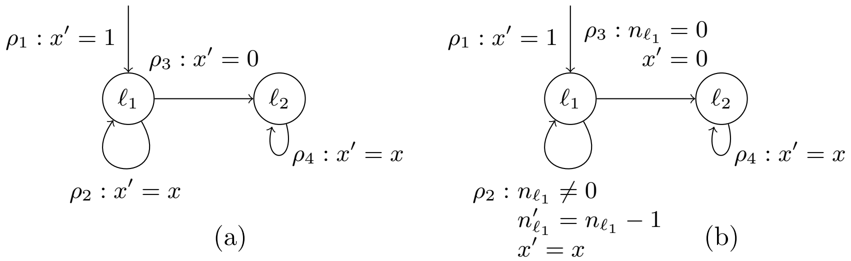

Fig. 1: (a) The control-flow graph of a program for which we wish to prove the CTL^* property $\mathsf{EFG}\ x = 1$. (b) The control-flow graph after calling Determinize, it includes the prophecy variable n_{ℓ_1} corresponding to the nondeterministic relation pair (ρ_2, ρ_3).

is, nondeterministic decisions regarding which paths are taken would be determined by *prophecy variables*, which determine future outcomes of the program execution, and their values [1]. Recall that relation pairs are distinguished if they are not part of the same strongly connected subgraph.

QuantElim acquires the proper set of states that satisfy a formula which has been verified over a determinized program. This allows for the path quantification present within a CTL^* formula, that is, whether all paths (or some paths) starting from a state satisfy a path formula. When a CTL^* formula of the form $\theta ::= \mathsf{A}\psi \mid \mathsf{E}\psi$ is reached after acquiring a set of states satisfying ψ, θ is verified on the same determinized program used for ψ. We then must use quantifier elimination to acquire the proper set of states that satisfy θ, thus quantifying the assertions over the values of the prophecy variables. If the formula is of the form $\mathsf{A}\psi$, we universally quantify the prophecy variables appearing in the set of states that satisfy $\mathsf{A}\psi$. If the formula is of the form $\mathsf{E}\psi$, we existentially quantify the prophecy variables.

Example. Consider the program in Fig. 1(a) and the property $\mathsf{EFG}\ x = 1$ stating that there exists a possible future where $x = 1$ will eventually become true and stay true. This is a system stabilization property which can only be expressed in CTL^*. We begin by identifying that $\mathsf{G}\ x = 1$ is a path formula, and thus use Approximate to return the over-approximated state formula $\mathsf{AG}\ x = 1$. We then initiate a CTL model checking task where we seek a set of states a_{G} such that $\mathsf{EF}a_{\mathsf{G}}$ holds, and for every state s such that $s \models a_{\mathsf{G}}$ we have $s \models \mathsf{AG}\ x = 1$.

Our formula would now only be valid if we can find a set of states that are eventually reached in a possible future from the program's initial states such that $\mathsf{AG}\ x = 1$ holds. However, no such set of states exists as the nondeterministic choice from ℓ_1 to ρ_2 and ρ_3 does not allow us to determine if we will eventually leave the loop or not. That is, there exists no set of states which can exemplify the infinite branching possibilities of leaving ρ_2 to possibly reaching ρ_3 or remaining in ρ_2 forever. In order to reason about the original sub-formula $\mathsf{G}\ x = 1$, we must be observing sets of paths, not states. Given that we over-approximated our formula in a way that allows us to only reason about states, we thus symbolically determinize the program to simultaneously simulate all possible related paths through the control flow graph and try to separate them to originate from distinct states in the program.

Our procedure Determinize would then return a new partially symbolically determinized system in which a newly introduced prophecy variable, named n_{ℓ_1} in Fig. 1(b), is associated with the relation pair (ρ_2, ρ_3), and is used to make predictions about the occurrences of relations ρ_2 and ρ_3. Recall that relation pairs correspond to pairs of nondeterministic transitions, one remaining in a SCS and the other leaving the same SCS. In this case, ρ_3 is indeed disjoint from the strongly connected subgraph of ℓ_1.

Given that we initialize n_{ℓ_1} to a nondeterministic value, for every path in the program, a positive concrete number chosen at the nondeterministic assignment predicts the number of instances that transition ρ_2 is visited before transitioning to ρ_3. That is, we remain in ρ_2 until $n_{\ell_1} = 0$, with n_{ℓ_1} being decremented at each passage through the loop. Once we terminate the loop, the prophecy variable is nondeterministically reset (for the case that we return to the same loop again). A negative assignment to n_{ℓ_1} denotes remaining in ρ_2 forever, or non-termination.

We can now utilize an existing CTL model-checker to return an assertion characterizing the states in which $\mathsf{G}\ x = 1$ holds by verifying the determinized program, denoted by P_D, using the over-approximated CTL formula $\mathsf{AG}\ x = 1$. The assertion $a_{\mathsf{G}} = (\ell_1 \wedge n_{\ell_1} < 0)$ is returned, and we proceed by replacing the sub-formula with its assertion in the original CTL^* formula, resulting in $\mathsf{EF}a_{\mathsf{G}}$. To verify the outermost CTL^* formula, EF, note that syntactically this is a readily acceptable CTL formula. However, we cannot simply use a CTL model checker as the path quantifier E exists within a larger relation context reasoning about paths given the inner formula FG. We thus must use the CTL model-checker to verify $\mathsf{EF}a_{\mathsf{G}}$ over the same determinized program previously generated.

Our procedure returns with the same precondition $(\ell_1 \wedge n_{\ell_1} < 0)$. We then use quantifier elimination to existentially quantify out all introduced prophecy variables. The existential quantification corresponds to searching for some path (or paths) that satisfy the path formula. Thus, if there is a state s in the original program, and some value of the prophecy variables v such that *all* paths from the combined state $(s, n_{\ell_1} = v)$ in P_D satisfy the path formula then clearly, these paths give us a sufficient proof to conclude that $\mathsf{EFG}\ x = 1$ holds from s in P.

4 Checking CTL^* Formulae

In this section, we describe the details of our CTL^* model checking procedure ProveCTL*. We first define the procedures utilized by ProveCTL*, namely Determinize and Approximate, followed by our model checking procedure and its utilization of QuantElim.

Determinize. The procedure Determinize constructs a form of partially symbolically determinized program over relation pairs that characterize instances of branching nondeterminism. We present our procedure in Fig. 2(a), where a program P is given and a partially determinized program P_D, contingent upon nondeterministic relation pairs, is returned. Ultimately, Determinize is designed to allow proof tools for branching-time logic state formulae to be used to reason about path formulae.

```
1  Let $\textsc{Determinize}(P) : \mathit{program} =$
2    $P_D = P$
3    $\textsc{Synth} = [\,]$
4    $(\mathcal{L}_D, E_D, \mathsf{Vars}_D) = P_D$
5    $C = \textsc{CyclePoints}(P)$
6    foreach $(\ell, \rho, \ell') \in E_D$ do
7      $G = \textsc{MinSCS}(P, C, \ell) \in \mathrm{SCS}(P, C)$
8      if $G \neq \emptyset \wedge \textsc{MinSCS}(P, C, \ell') \neq G$ then
9        $\textsc{Synth} = \ell :: \textsc{Synth}$
10   done
11   foreach $(\ell, \rho, \ell') \in E_D$ do
12     if $\ell \in \textsc{Synth}$ then
13       $\mathsf{Vars}_D = \mathsf{Vars}_D \cup n_\ell \in \mathbb{Z}$
14       if $\ell' \in \textsc{MinSCS}(P, C, \ell)$ then
15         $\rho = \rho \wedge (n_\ell \neq 0) \wedge (n'_\ell = n_\ell - 1)$
16       else
17         $\rho = \rho \wedge (n_\ell = 0)$
18   done
19   return $P_D$
```

(a)

```
1  Let $\textsc{Approximate}(\psi, a_{\theta'_1}, a_{\theta'_2}) : \varphi =$
2    match $(\psi)$ with
3      | $\mathsf{F}\theta'_1 \to \mathsf{AF}a_{\theta'_1}$
4      | $\mathsf{G}\theta'_1 \to \mathsf{AG}a_{\theta'_1}$
5      | $\mathsf{X}\theta'_1 \to \mathsf{AX}a_{\theta'_1}$
6      | $\theta'_1 \,\mathsf{W}\, \theta'_2 \to \mathsf{A}a_{\theta'_1} \,\mathsf{W}\, a_{\theta'_2}$
7      | $\theta'_1 \,\mathsf{U}\, \theta'_2 \to \mathsf{A}a_{\theta'_1} \,\mathsf{U}\, a_{\theta'_2}$
8      | $\theta'_1 \wedge \theta'_2 \to a_{\theta'_1} \wedge a_{\theta'_2}$
9      | $\theta'_1 \vee \theta'_2 \to a_{\theta'_1} \vee a_{\theta'_2}$
```

(b)

```
1  Let $\textsc{Verify}(\theta, P) : \mathit{bool} =$
2      $(\mathcal{L}, E, \mathsf{Vars}) = P$
3      $P_D = \textsc{Determinize}(P)$
4      $(a, \_) = \textsc{ProveCTL}^*(\theta, P, P_D)$
5      return $\forall(\ell_0, \rho, \ell) \in E \;\forall s \,.\, (s, s) \models \rho \Rightarrow a$
```

(c)

```
1  Let $\textsc{QuantElim}(a, \varphi) : AP =$
2    $a_{\mathsf{EG}} = \mathrm{CTL}(P_D, \mathsf{EG}\ \textsc{True})$
3    match $(\varphi)$ with
4      | $\mathsf{A}\psi \to \neg QE(\exists n_{\ell \in \mathcal{L}} . a_{\mathsf{EG}} \wedge \neg a)$
5      | $\mathsf{E}\psi \to QE(\exists n_{\ell \in \mathcal{L}} . a_{\mathsf{EG}} \wedge a)$
```

(d)

Fig. 2: (a) Determinize identifies relation pairs and constructs a symbolically determinized program over them. (b) Approximate produces a syntactic conversion from a path formula to its corresponding over-approximation in ACTL. (c) Verify wraps ProveCTL* and then checks all initial states. (d) QuantElim applies quantifier elimination in order to convert path characterization to state characterization restricting attention to states from which an infinite path exists.

We begin by finding a sufficient set of relation pairs to symbolically determinize the program to one which has the same set of paths as the original. These relations are distinguished if there exist two nondeterministic relations stemming from the same location and yet are not part of the same strongly-connected subgraph. Our procedure thus begins by iterating over the set of a program's edges, $(\ell, \rho, \ell') \in E$ on line 6. We identify whether or not $\ell \in C$ given that $G = \textsc{MinSCS}(P, C, \ell)$ and $G \neq \emptyset$ on lines 7 and 8. If from some location ℓ, where $G = \textsc{MinSCS}(P, C, \ell)$, there is an edge to ℓ' such that $\textsc{MinSCS}(P, C, \ell')$ is not equivalent to G, we can conclude that the transition from ℓ to ℓ' leaves the SCS of ℓ. We only desire that ℓ and ℓ' be elements of the most minimal SCS as such an edge eludes to the nondeterministic decision point where a transition diverted from remaining within an SCS. This nondeterministic point is key to the identification of where determinization must occur to facilitate the application of state-based reasoning to path-based reasoning for given a program P.

If the strongly connected subgraphs of ℓ and ℓ' do differ, we add ℓ to Synth, a list which tracks locations with nondeterministic points. For every such location, we identify a relation pair corresponding to the decision of either remaining in the same SCS, or leaving it. After finding all possible elements of Synth, on line

```
Let rec $\textsc{ProveCTL}^*(\theta, P, P_D) : (formula, bool) =$
  $(\mathcal{L}, E, \mathsf{Vars}) = P$
  match $(\theta)$ with
  | $\varphi : stateformula \;\rightarrow$
     match $(\varphi)$ with
      | $\alpha \;\rightarrow a_\theta = \alpha$; $\textsc{Path} = \textsc{False}$
      | $\theta'_1 \wedge \theta'_2$ | $\theta'_1 \vee \theta'_2$ | $\mathsf{E}\theta'_1 \,\mathsf{U}\, \theta'_2$ | $\mathsf{A}\theta'_1 \,\mathsf{W}\, \theta'_2$
      | $\mathsf{E}\theta'_1 \wedge \theta'_2$ | $\mathsf{E}\theta'_1 \vee \theta'_2$ | $\mathsf{A}\theta'_1 \wedge \theta'_2$ | $\mathsf{A}\theta'_1 \vee \theta'_2 \rightarrow$
          $(a_{\theta'_1}, \textsc{Path}_1) = \textsc{ProveCTL}^*(\theta'_1, P, P_D)$
          $(a_{\theta'_2}, \textsc{Path}_2) = \textsc{ProveCTL}^*(\theta'_2, P, P_D)$
      | $\mathsf{AF}\theta'$ | $\mathsf{AG}\theta'$ | $\mathsf{AX}\theta'$ | $\mathsf{EF}\theta'$ | $\mathsf{EG}\theta'$ | $\mathsf{EX}\theta' \rightarrow$
          $(a_{\theta'_1}, \textsc{Path}_1) = \textsc{ProveCTL}^*(\theta', P, P_D)$
          $\textsc{Path}_2 = \textsc{False}$
     if $\varphi \neq \alpha$ then
        $\varphi' = \textsc{Replace}(\psi, a_{\theta'_1}, a_{\theta'_2})$
        if $\textsc{Path}_1 \vee \textsc{Path}_2$ then
           $a_\theta = \textsc{QuantElim}(\mathrm{CTL}(P_D, \varphi'), \varphi)$
           $\textsc{Path} = \textsc{false}$
        else
           $a_\theta = \mathrm{CTL}(P, \varphi')$
           $\textsc{Path} = \textsc{False}$
  | $\psi : pathformula \;\rightarrow$
     match $(\psi)$ with
     | $\theta'_1 \wedge \theta'_2$ | $\theta'_1 \vee \theta'_2$ | $\theta'_1 \,\mathsf{U}\, \theta'_2$ | $\theta'_1 \,\mathsf{W}\, \theta'_2 \rightarrow$
          $(a_{\theta'_1}, \_) = \textsc{ProveCTL}^*(\theta'_1, P, P_D)$
          $(a_{\theta'_2}, \_) = \textsc{ProveCTL}^*(\theta'_2, P, P_D)$
     | $\mathsf{F}\theta'$ | $\mathsf{G}\theta'$ | $\mathsf{X}\theta' \rightarrow$
          $(a_{\theta'_1}, \_) = \textsc{ProveCTL}^*(\theta', P, P_D)$
     $\psi' = \textsc{Approximate}(\psi, a_{\theta'_1}, a_{\theta'_2})$
     $a_\theta = \mathrm{CTL}(P_D, \psi')$
     $\textsc{Path} = \textsc{true}$
  $(a_\theta, \textsc{Path})$
```

Fig. 3: Our recursive CTL^* verification procedure employs an existing CTL model checker and uses our procedures Approximate and QuantElim. It expects a CTL^* property θ, a program P, and its determinized version P_D as parameters. An assertion characterizing the states in which θ holds is returned along with a boolean value indicating whether the formula checked was a path formula (and hence approximated).

11 we iterate over the program edges, and for each relation pair encountered we introduce a new prophecy variable to predict the future outcome of the decision. Indeed, our motivation is to identify nondeterministic points so we can symbolically simulate all possible branching paths through a program, yet decisions regarding which paths are taken are determined by prophecy variables and their values. Information regarding different paths is now stored in the state of the modified program. This allows for a correspondence such that the verification path formulae can be reduced to the verification of ACTL formulae.

When an edge $(\ell, \rho, \ell') \in E$ is reached containing $\ell \in \textsc{Synth}$, a prophecy variable $n_\ell \in \mathbb{Z}$ is added to the set of program variables Vars at line 13. If ℓ' is contained within $\textsc{MinSCS}(P, C, \ell)$, we constrain ρ by requiring that $n_\ell > 0$, and then decrement n_ℓ. If ℓ' is not contained within $\textsc{MinSCS}(P, C, \ell)$, we constrain ρ by $n_\ell = 0$, and n'_ℓ remains unconstrained, entailing a reset to a nondeterministic integer. The nondeterministic decision of the number of times a cycle is passed through is thus now determined by the prophecy variable n_ℓ. In the case that $n_\ell < 0$, this rule corresponds to behaviors where every visit to ℓ is followed by a successor in the same SCS (i.e., the computation always remains in the SCS of ℓ). The nondeterminism within a transition relation is thus either determined at initialization by the initial choice of values for n_ℓ or else later in a path by choosing new nondeterministic values for n_ℓ.

We show that the determinization maintains the set of paths in the original program and the prophecy variables introduced merely trade nondeterminism in the transition relation for a larger, nondeterministic state space.

Theorem 1. *For every path π in P there is a path π' in P_D such that $\pi' \Downarrow_{\mathsf{Vars}} = \pi$. Furthermore, for every path π' in P_D it holds that $\pi' \Downarrow_{\mathsf{Vars}}$ is a path in P.*
Proof. *See Appendix A.*

Approximate. In Fig. 2(b), we present a syntactic conversion from pure linear-time formulae in CTL^*, that is LTL, to a corresponding over-approximation in ACTL. Our procedure is given a path formula ψ and two atomic preconditions, $a_{\theta'_1}$ and $a_{\theta'_2}$, corresponding to satisfaction of the nested CTL^* formulae which appear within ψ. The precondition $a_{\theta'_2}$ is a conditional parameter utilized only when LTL formulae requiring two properties (e.g. W, U, $\wedge$, $\vee$) are given. Due to the recursive nature of PROVECTL*, presented in the next section, these preconditions would have already been priorly generated.

On lines 3 – 7, we instrument a universal path quantifier A preceding the appropriate temporal operators. Not only so, but the sub-formulae θ'_1 and θ'_2 are replaced with their corresponding preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$, respectively. This aligns with how PROVECTL* will recursively iterate over each inner sub-formula followed by search for the preconditions of the outer sub-formulae dependent on it. Replacing a path formula by its CTL approximation indeed is sound in the sense that if the modified formula holds then the original holds as well.

Theorem 2. *For every program P, a state (ℓ, f), and a path formula ψ, if $P, (\ell, f) \models$ APPROXIMATE(ψ) then $P, (\ell, f) \models \mathsf{A}\psi$.*
Proof. *See Appendix A.*

Theorem 2 does not consider existential path quantification. Recall that in order to conclude that the CTL^* formula $P, s \models \mathsf{E}\psi$ for some path formula ψ, we require that there is some value v of the prophecy variables such that $P_D, (s, v) \models \mathsf{A}\psi$. This means that when restricting attention to a certain set of paths that start in a state s (those that match the valuation v for prophecy variables), *all* paths in the set satisfy the formula ψ. Clearly, this satisfies the requirement that there is some path that satisfies the formula.

4.1 ProveCTL*

In this section, we present our main CTL^* verification procedure. Fig. 2(c) depicts VERIFY, which wraps the main procedure PROVECTL*, shown in Fig. 3. We then generate a determinized copy of the program, P_D, using the aforementioned procedure DETERMINIZE. This program is then passed into PROVECTL* along with the original program P and a CTL^* property θ. PROVECTL* then returns an assertion a, characterizing the states in which θ holds. The second argument returned is disregarded, indicated by "_", as it is only used within the recursive calls of PROVECTL*. When PROVECTL* returns to VERIFY, it is only necessary to check if the precondition a is satisfied by the initial states of the program.

In order to synthesize a precondition for a CTL^* property θ, we first recursively accumulate the preconditions generated when considering the sub-formulae of θ at lines 9, 10, 12, 25, 26, and 28. That is, for each sub-formula θ, we produce a precondition a_θ that ensures its satisfaction. We note that the precondition of an atomic proposition α is the proposition itself. A given CTL^* formula is then deconstructed to differentiate between state and path sub-formulae, as the crux of verifying CTL^* formulae lies within identifying the interplay between

the arbitrary nesting of path and state formulae. On line 3, if θ can be identified as a state formula φ, we carry out the set of actions on lines 4 – 21. If θ is identified as a path formula ψ, we then we carry set of actions on lines 22 – 31.
Verifying path formulae. When a path formula ψ is reached, we begin by over-approximating the path formula by syntactically converting it to the universal subset of branching-time logic (ACTL) using the procedure Approximate. Recall that the preconditions generated when considering the sub-formula(e) of ψ at lines 25, 26, and 28 will be utilized by Approximate to replace θ'_1 and θ'_2 with their corresponding preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$, respectively. On line 29, Approximate would then return a corresponding state formula ψ' where a universal path quantifier precedes every temporal operator within ψ.

A precondition for the newly attained ACTL formula ψ' can now be acquired via existing CTL model checkers which return an assertion characterizing the states in which ψ' holds. Existing tools which support this functionality include [3] and [10]. In our tool prototype, we build upon the latter. Recall that a precondition for a path formula requires more than a precondition for the corresponding state formula, as ψ' is merely an over-approximation. We thus must utilize the provided determinized program P_D when employing a CTL model checker rather than the original program P, as shown on line 30. The assertion a_θ is then returned characterizing the sets of states in which θ holds.

Recall that P_D leads to better correspondence between ψ and ψ'. That is, we find a sufficient set of relation pairs which determinize the program to one which has the same set of paths as the original, yet decisions regarding which paths are taken are determined by introduced prophecy variables and their values, allowing us to reduce path-based reasoning to state-based reasoning.

Finally, on line 31, we set the boolean flag Path to true. This flag is the second argument to be returned by ProveCTL*. It indicates to the caller that the result a_θ returned by the recursive call is approximated. The value of Path is used for deciding whether to use a_θ as is or modify it (in the case that the verified sub-formula is a state or a path formula, respectively), admitting a sound interaction between state and path formulae.
Verifying state formulae. In the case that a state formula φ is reached, we partition the state sub-formulae by the syntax of CTL as shown on lines 6 – 8 and 11. This allows us to not only utilize existing CTL model checkers, but to also eliminate the redundant verification of a temporal operator, when it is already be preceded by a path quantifier. As a side effect of partitioning φ in such a way, a path formula ψ will always be in the form of a pure linear-time path formula, that is, LTL. This particular deconstruction of a CTL* formula is what allows us to identify the intricate interplay between path and state formulae.

We begin by recursively generating preconditions when considering the sub-formula(e) of φ at lines 9, 10, and 12. These preconditions will then be utilized by the procedure Replace on line 15. Replace substitutes θ'_1 and θ'_2 with their corresponding preconditions $a_{\theta'_1}$ and $a_{\theta'_2}$, respectively, and returns a new state formula φ'. Preconditions for branching-time logic state formulae can be acquired via existing CTL model checkers. However, in order to allow for the path quantification present within a CTL^* formula to range over path formulae, we must consider whether all or some paths starting from a particular state satisfy a path formula. This is required in the case that the immediate inner sub-formula is a pure linear-time path formula, which is identified by the aforementioned boolean flag PATH given the partitioning of θ. The role of PATH is to track if a sub-formula of the current formula is a path formula. That is, PATH indicates that the path quantifier exists within the context of verifying a path formula, and not a branching-time state formula. Thus, it must be verified using P_D, yet the set of states of P_D that characterize it actually represents a set of paths. This set of paths must be collapsed later to a characterization of the set of states of P where the (state) formula holds. This is the key to allowing the interplay between state and path formulae.

The procedure QUANTELIM, presented in Fig. 2(d), which converts path characterization to state characterization, is thus executed at line 17. QUANTELIM takes in the assertion a returned from calling a CTL model checker on the determinized program P_D and the partitioned CTL formula φ', as well as the original formula φ. We then quantify the assertions over the values of the prophecy variables. If φ is a universal CTL formula, we universally quantify the prophecy variables appearing in the set of states that satisfy φ on line 4 in Fig. 2(d). If φ is an existential CTL formula, we existentially quantify the prophecy variables on line 5. Predictions of the prophecy variables may lead to finite paths to appear in the program, thus quantification must be restricted to states for which there does exist a prophecy value leading to infinite paths. Hence, on line 2 we acquire the precondition a_{EG} satisfying the CTL formula entailing nontermination, that is EG TRUE for P_D. The precondition a_{EG} is then conjuncted with a to ensure that the quantification of prophecy variables does not include finite paths generated due to invalid predictions of the prophecy variables. This is done according to the polarity of the quantification (universal or existential). The assertion a_θ is then returned by QUANTELIM characterizing the set of states in which θ holds.

In the case that PATH is false, the most immediate inner sub-formula would then be a state formula. This indicates that we can indeed use a CTL model checker using φ' and the original program P, as demonstrated on line 20. Upon the return of PROVECTL* to its caller VERIFY, a_θ will contain the precondition for the most outer temporal property of the original CTL^* formula θ. Now it is only necessary to check if the precondition a_θ is satisfied by the initial states of the program to complete the verification of our CTL^* formula. Finally, PATH is set to false, in order to carry out the above procedure again when necessary.

Theorem 3. *If* VERIFY(θ, P) *returns true then* $P \models \theta$.
Proof. *See Appendix A.*

We note that the implication in Theorem 3 is only in one direction. That is, failing to prove that a property holds does not implicate that its negation holds (though this might be proved by negating the formula, converting it to negation normal form, and running our procedure on it). This incompleteness stems from the over-approximation of path formulae by a corresponding ACTL formulae, as although this over-approximation is checked over P_D, P_D does not determinize all paths. It is impossible to completely determinize a program as this requires uncountable branching (in the choice of prophecy variables). Countable nondeterminism is not a sufficient technique in the context of nondeterministic nested determinization of programs. For example, suppose that the prophecy variable value entails that an external loop does not terminate. Now consider all possible options for number of repetitions of the internal loop. In order to have a completely deterministic program, we must prophesize an infinite sequence of finite natural numbers. The number of such possible infinite sequences is uncountable.

5 Evaluation

Program	**LoC**	**Property**	**Time(s)**	**Res.**
OS frag. 1	393	$\mathsf{AG}((\mathsf{EG}(\texttt{phi\_io\_compl} \leq 0)) \vee (\mathsf{EFG}(\texttt{phi\_nSUC\_ret} > 0))))$	32.0	×
OS frag. 1	393	$\mathsf{EF}((\mathsf{AF}(\texttt{phi\_io\_compl} > 0)) \wedge (\mathsf{AGF}(\texttt{phi\_nSUC\_ret} \leq 0))))$	13.2	✓
OS frag. 2	380	$\mathsf{EFG}((\texttt{keA} \leq 0 \wedge (\mathsf{AG}\ \texttt{keR} = 0)))$	28.3	✓
OS frag. 2	380	$\mathsf{EFG}((\texttt{keA} \leq 0 \vee (\mathsf{EF}\ \texttt{keR} = 1)))$	16.5	✓
OS frag. 3	50	$\mathsf{EF}(\texttt{PPBlockInits} > 0 \wedge (((\mathsf{EFG}\ \texttt{IoCreateDevice} = 0)$ $\vee (\mathsf{AGF}\ \texttt{status} = 1)) \wedge (\mathsf{EG}\ \texttt{PPBunlockInits} \leq 0)))$	10.4	✓
PgSQL arch 1	106	$\mathsf{EFG}(\texttt{tt} > 0 \vee (\mathsf{AF}\ \texttt{wakend} = 0))$	1.5	×
PgSQL arch 1	106	$\mathsf{AGF}(\texttt{tt} \leq 0 \wedge (\mathsf{EG}\ \texttt{wakend} \neq 0))$	3.8	✓
PgSQL arch 1	106	$\mathsf{EFG}(\texttt{wakend} = 1 \wedge (\mathsf{EGF}\ \texttt{wakend} = 0))$	18.3	✓
PgSQL arch 1	106	$\mathsf{EGF}(\mathsf{AG}\ \texttt{wakend} = 1)$	10.3	✓
PgSQL arch 1	106	$\mathsf{AFG}(\mathsf{EF}\ \texttt{wakend} = 0)$	1.5	×
PgSQL arch 2	100	$\mathsf{AGF}\ \texttt{wakend} = 1$	1.4	✓
PgSQL arch 2	100	$\mathsf{EFG}\ \texttt{wakend} = 0$	0.5	×
Bench 1	12	$\mathsf{EFG}(\texttt{x} = 1 \wedge (\mathsf{EG}\ \texttt{y} = 0))$	1.0	✓
Bench 2	12	$\mathsf{EGF}\ \texttt{x} > 0$	0.1	✓
Bench 3	12	$\mathsf{AFG}\ \texttt{x} = 1$	0.1	✓
Bench 4	10	$\mathsf{AG}((\mathsf{EFG}\ \texttt{y} = 1) \wedge (\mathsf{EF}\ \texttt{x} \geq \texttt{t}))$	0.5	×
Bench 5	10	$\mathsf{AG}(\texttt{x} = 0\ \mathsf{U}\ \texttt{b} = 0)$	T/O	–
Bench 6	8	$\mathsf{AG}((\mathsf{EFG}\ \texttt{x} = 0) \wedge (\mathsf{EF}\ \texttt{x} = 20))$	0.1	✓
Bench 7	6	$(\mathsf{EFG}\texttt{x} = 0) \wedge (\mathsf{EFG}\texttt{y} = 1)$	0.5	×
Bench 8	6	$\mathsf{AG}((\mathsf{AFG}\ \texttt{x} = 0) \vee (\mathsf{AFG}\texttt{x} = 1))$	0.5	✓

Fig. 4: Experimental evaluations of infinite-state programs drawn from the Windows OS, PgSQL, and 8 toy examples. There are no competing tools available for comparison.

In this section we discuss the results of our experiments with an implementation of the procedure from Fig. 2(c). Our implementation[2] is built as an extension to the open source project T2, which uses a safety prover similar to Impact [23] alongside previously published techniques for discovering ranking functions, etc. [24, 8] to prove both liveness and safety properties. The tool was

[2] The source-code of our implementation and our benchmarks are available under the MIT open-source license at `https://github.com/hkhlaaf/T2/tree/T2Star`.

executed on an Intel x64-based 2.8 GHz single-core processor. The format in which we interpret and parse a program's commands can be found in [10].

We have drawn out a set of CTL^* problems from industrial code bases. Examples were taken from the I/O subsystems of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the Apache web server. CTL^* allows us to express "possibility" properties, such as the viability of a system, stating that any reachable state can spawn a fair computation. Additionally, we demonstrate that we can now verify properties involving existential system stabilization, stating that an event can eventually become true and stay true from any reachable state. For example, "OS frag. 1", "OS frag. 3", "PgSQL arch 1", and "Bench 2" are verified using said properties, described in detail in Section 1. We also include a few toy examples to further demonstrate further expressiveness of CTL^* and its usefulness in verifying programs.

Given that our benchmarks tackle infinite-state programs, the only existing automated tool for verifying CTL^* in the finite-state setting [17] is not applicable. In Figure 4 we display the results of our benchmarks. For each program and its corresponding CTL^* property to be verified, we display the number of lines of code (LoC), and report the time it took to verify a CTL^* property (Time column) in seconds. We provide a **"Res."** column which indicates the results of our tool. A ✓ indicates that the tool was able to verify the property. Likewise, an × indicates that the tool failed to prove the property. The symbol "–" in the result column indicates that a result was not determined due to a timeout. A timeout or memory exception is indicated by $\mathsf{T/O}$. A timeout is triggered if verification of an experiment exceeds 3000 seconds. Note that in various cases, we verify the same program using a CTL^* property and its negation. Our tool thus allows us to prove each of the properties as well as disprove each of their negations.

Our experiments demonstrate the practical viability of our approach. Our runtimes show that our tool runs well within the range of performance previously exhibited by specialized tools such as as [9, 11, 7, 3, 10], which can only verify significantly less expressive properties over infinite-state programs. Our tool has successfully both verified and invalidated CTL^* properties corresponding to their expected results for all but one of the benchmarks. This is due to the aforementioned limitation, that is, our countable nondeterministic determinization technique is not complete.

6 Concluding Remarks

We have introduced the first-known fully automatic method capable of proving CTL^* of infinite-state (integer) programs. This allows us, for the first time ever, to automatically verify properties of programs that mix branching-time and linear-time temporal operators. We have developed an implementation capable of automatically proving properties of programs that no tool could previously prove. The method underlying our tool is one that uses a symbolic representation capable of facilitating reasoning about the interaction between sets of states and sets of paths.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.
2. T. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL '14*, pages 221–233. ACM, 2014.
3. T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV'13*, pages 869–882. Springer, 2013.
4. N. S. Bjørner, A. Browne, M. A. Colón, B. Finkbeiner, Z. Manna, H. B. Sipma, and T. E. Uribe. Verifying temporal properties of reactive systems: A step tutorial. *Form. Methods Syst. Des.*, 16(3):227–270, 2000.
5. E. Bodden. A lightweight ltl runtime verification tool for java. In *OOPSLA '04*, pages 306–307. ACM, 2004.
6. B. Cook, H. Khlaaf, and N. Piterman. Fairness for infinite-state systems. In *TACAS '15*, Springer, 2015. To appear.
7. B. Cook and E. Koskinen. Reasoning about nondeterminism in programs. In *PLDI'13*, pages 219–230. ACM, 2013.
8. B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS'13*, LNCS, pages 47–61. Springer, 2013.
9. B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL'07*, pages 265–276. ACM, 2007.
10. B. Cook, H. Khlaaf, and N. Piterman. Faster temporal reasoning for infinite-state programs. In *FMCAD '14*, pages 16:75–16:82. FMCAD Inc, 2014.
11. B. Cook and E. Koskinen. Making prophecies with decision predicates. In *POPL'11*, pages 399–410. ACM, 2011.
12. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.
13. E. A. Emerson and J. Y. Halpern. "sometimes" and "not never"; revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.
14. E. A. Emerson and C. S. Jutla. The complexity of tree automata and logics of programs. *SIAM J. Comput.*, 29(1):132–158, 1999.
15. E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.
16. E. A. Emerson and A. P. Sistla. Deciding branching time logic. In *STOC '84*, pages 14–24. ACM, 1984.
17. A. Griffault and A. Vincent. The Mec 5 model-checker. In *CAV'04*, LNCS, pages 488–491. springer, 2004.
18. A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. *SIGPLAN Not.*, 43:147–158, January 2008.
19. Y. Kesten and A. Pnueli. A compositional approach to ctl* verification. *Theor. Comput. Sci.*, 331(2-3):397–428, 2005.
20. L. Lamport. "sometime" is sometimes "not never": On the temporal logic of programs. In *POPL '80*, pages 174–185. ACM, 1980.
21. S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS'07*, pages 419–436. Springer, 2007.
22. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*, volume 2. Springer, 1995.

23. K. McMillan. Lazy abstraction with interpolants. In *CAV'06*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.
24. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41, Turku, Finland, 2004. IEEE.
25. M. Reynolds. An axiomatization of full computation tree logic. *The Journal of Symbolic Logic*, 66(3):pp. 1011–1057, 2001.
26. F. Song and T. Touili. Pushdown model checking for malware detection. In *TACAS'12*, pages 607–610. ACM, 2012.

A Proofs

Theorem 1. *For every path π in P there is a path π' in P_D such that $\pi'\Downarrow_{\mathsf{Vars}} = \pi$. Furthermore, for every path π' in P_D it holds that $\pi'\Downarrow_{\mathsf{Vars}}$ is a path in P.*

Proof. Consider a path π in P where $\pi = (\ell_0, f_0)$, $(\ell_1, f_1), \ldots$. Consider a location ℓ_j, a SCS G_j such that $G_j = \textsc{MinSCS}(P, C, \ell_j)$, and the variable n_{l_j}. We can annotate each pair (ℓ_i, f_i) in π by the number of expected future visits to G_j. We call a transition (ℓ, ρ, ℓ') a *reset transition* for n_{l_j} if $\ell \in G_j$ and $\ell' \notin G_j$ or if $\ell = \ell_I$. Notice that in P_D, the transition (ℓ, ρ, ℓ') is conjuncted to $n_{l_j} = 0$. This leaves the value of n'_{l_j} unconstrained, assigning it an arbitrary value once such a transition is taken. We call a transition (ℓ, ρ, ℓ') an *internal transition* for n_{l_j} if $\ell \in G_j$, $\ell' \in G_j$ and there is some $\ell'' \notin G_j$ and a transition (ℓ, ρ', ℓ''). Notice that in P_D the transition (ℓ, ρ, ℓ') is conjuncted to $n'_{l_j} = n_{l_j} - 1$. Also, in P_D every transition that is neither reset nor internal for n_{l_j} is conjuncted to $n'_{l_j} = n_{l_j}$. It follows that for every $i \geq 0$ the number of internal transitions for n_{l_j} that appear until a reset transition is well-defined (and may be infinity). Clearly, this annotation also matches the transition in P_D.

It follows that by adding an appropriate annotation for every n_l that is added to P_D, we get a path in P_D whose projection on Vars is exactly that of path π.

Consider an infinite path π' in P_D. Now consider a pair of states $((\ell, (f, v)), (\ell', (f', v'))$ appearing in π', where v and v' are the assignments to the prophecy variables appearing in P_D. By definition, there is a transition (ℓ, ρ', ℓ') in P_D such that $((\ell, (f, v)), (\ell', (f', v')) \models \rho'$. However, $\rho' = \rho \wedge \xi$, where ρ is an assertion over Vars and ξ is the assertion over the prophecy variables. It then must be the case that $(f, f') \models \rho$. It follows that $\pi = \pi'\Downarrow_{\mathsf{Vars}}$ is a path in P.

Theorem 2. *For every program P, a state (ℓ, f), and a path formula ψ, if $P, (\ell, f) \models$ $\textsc{Approximate}(\psi)$ then $P, (\ell, f) \models A\psi$.*

Proof. The proof proceeds by induction on the structure of the path formula ψ. For propositions and boolean combinations of simpler formulae, the proof is immediate.

- Suppose that $\psi = \mathsf{G}\psi'$. Then, $\textsc{Approximate}(\psi)$ is $\mathsf{AG}(\textsc{Approximate}(\psi'))$. Suppose that $(\ell, f) \models$ $\textsc{Approximate}(\psi)$ but $(\ell, f) \not\models \mathsf{A}\psi$. Then, there is a path π starting in (ℓ, f)

such that π does not satisfy $\mathsf{G}\psi'$. It follows that there is a suffix π' of π that does not satisfy ψ'. Let (ℓ', f') be the first state in π'. However, by assumption, $(\ell', f') \models$ APPROXIMATE(ψ'). This contradicts the induction hypothesis.

- Suppose that $\psi = \mathsf{F}\psi'$. Then, APPROXIMATE(ψ) is $\mathsf{AF}($APPROXIMATE$(\psi'))$. Suppose that $(\ell, f) \models$
APPROXIMATE(ψ) $(\ell, f) \not\models \mathsf{A}\psi$. Then, there is a path π starting in (ℓ, f) such that π does not satisfy $\mathsf{F}\psi'$. However, by $(\ell, f) \models$ APPROXIMATE(ψ), there is a suffix π' of π such that the first state (ℓ', f') in π' satisfies APPROXIMATE(ψ'). It follows that π' satisfies ψ' and that π satisfies $\mathsf{AF}\psi'$.
- The proofs for until and weak until are similar but take further corner cases into account.

Theorem 3. *If* VERIFY(θ, P) *returns true then* $P \models \theta$.

Proof. We show by induction on the number of path quantifiers in the CTL^* formula θ that the set of states computed as satisfying θ in line 14 of PROVECTL* is sound. That is, if a state (ℓ, f) is such that $(\ell, f) \models a_\theta$ then (ℓ, f) satisfies θ.

- Consider a state formula $\mathsf{A}\psi$, where ψ does not include further path quantifications. Suppose that $(\ell, f) \models a_\theta$. As a_θ is obtained from universal quantification of prophecy variables in P_D it follows that for every possible valuation v for the prophecy variables either $(\ell, (f, v))$ has no infinite paths starting from it or $(\ell, (f, v))$ satisfies APPROXIMATE(φ) in P_D. By Theorem 2 every path in P_D that starts in $(\ell, (f, v))$ satisfies ψ.
Consider a path π that starts in (ℓ, f) in P and Let $\sigma \cdot \pi$ be a computation of P for which π is a suffix. Then, by Theorem 1 there exists a computation $\sigma' \cdot \pi'$ of P_D such that $\sigma' \cdots \pi' \Downarrow_{\mathsf{Vars}} = \sigma \cdot \pi$. In particular, π' satisfies ψ as required.
- Consider a state formula $\mathsf{E}\psi$, where ψ does not include further path quantifications. Suppose that $(\ell, f) \models a_\theta$. As a_θ is obtained from existential quantification of prophecy variables in P_D it follows that there is a valuation v for the prophecy variables such that $(\ell, (f, v))$ satisfies APPROXIMATE(φ) in P_D. Furthermore, as $(\ell, (f, v))$ satisfies $\mathsf{EG}\mathit{true}$, there is some infinite path starting from it. By theorem 2 every path in P_D that starts in $(\ell, (f, v))$ satisfies ψ.
By construction of APPROXIMATE(θ) there exists an infinite path π' of P_D that starts in (ℓ, f). Let σ' be some prefix such that $\sigma' \cdot \pi'$ is a computation of P_D. It follows that $\sigma' \cdot \pi' \Downarrow_{\mathsf{Vars}}$ is a computation of P and that $\pi' \Downarrow_{\mathsf{Vars}}$ is the witness to (ℓ, f) satisfying $\mathsf{E}\psi$.
- In the case of a state formula θ that includes nesting of path quantifiers the proof proceeds as before. This part relies on the structure of θ being in negation normal form and the soundness of previous approximations $a_{\theta'}$ for every state sub-formula θ' of θ.